
Multiplicative LSTM for sequence modelling

Ben Krause¹, Liang Lu², Iain Murray¹, Steve Renals¹

¹University of Edinburgh, School of Informatics

²Toyota Technological Institute at Chicago

ben.krause@ed.ac.uk, ll@ttic.edu,
i.murray@ed.ac.uk, s.renals@ed.ac.uk

Abstract

This paper introduces multiplicative LSTM, a novel hybrid recurrent neural network architecture for sequence modelling that combines the long short-term memory (LSTM) and multiplicative recurrent neural network architectures. Multiplicative LSTM is motivated by its flexibility to have very different recurrent transition functions for each possible input, which we argue helps make it more expressive in autoregressive density estimation. We show empirically that multiplicative LSTM outperforms standard LSTM and deep variants for a range of character level modelling tasks. We also found that this improvement increases as the complexity of the task scales up. This model achieves a validation error of 1.20 bits/character on the Hutter prize dataset when combined with dynamic evaluation.

1 Introduction

Recurrent neural networks (RNNs) have proven to be powerful sequence density estimators that can use long contexts to make predictions. RNNs are in theory powerful enough to express sequential probability density functions of arbitrary complexity, but in practice, this is difficult to realize. Therefore, much work has gone into making RNNs more expressive and easier to fit.

Generative models of sequences can apply factorization via the product rule to perform density estimation of the elements of the sequence $x_{1:T} = \{x_1, \dots, x_T\}$, which is given by

$$P(x_1, \dots, x_T) = P(x_1)P(x_2|x_1)P(x_3|x_2, x_1) \dots P(x_T|x_1 \dots x_{T-1}). \quad (1)$$

The number of possible histories grows exponentially with the length of the sequence, requiring an approximation to be used in place of counts. Some models assume that the sequence has the Markov property, and only consider the n most recent sequence elements when predicting x_t . RNNs avoid assuming the Markov property by using a hidden state to summarize the past inputs. The hidden state vector h_t is updated recursively using the past hidden state vector h_{t-1} and the new input x_t as

$$h_t = \mathcal{F}(h_{t-1}, x_t), \quad (2)$$

where \mathcal{F} is a differentiable function with learnable parameters. In the vanilla RNN, \mathcal{F} multiplies its inputs by a matrix and squashes the result with a non-linear function such as a hyperbolic tangent. The updated hidden state vector is then used to predict a probability distribution over the next sequence element, using function \mathcal{G} . In the case where $x_{1:T}$ consists of mutually exclusive discrete outcomes, \mathcal{G} may apply a matrix multiplication followed by a softmax function:

$$P(x_{t+1}) = \mathcal{G}(h_t). \quad (3)$$

Generative RNNs are fully tractable density estimators, meaning that they can evaluate log-likelihoods of sequences exactly. They are also differentiable with respect to these log-likelihoods, allowing them to be trained to maximize the probability of training data using gradient descent based methods.

RNNs can be difficult to train due to the vanishing gradient problem (Bengio et al., 1994; Hochreiter et al., 2001), but advances such as the long short-term memory architecture (LSTM) (Hochreiter and Schmidhuber, 1997) have allowed RNNs to be successful generative sequence modellers. Despite their success, generative RNNs (as well as other conditional generative models) are known to have problems with recovering from mistakes when predicting sequences (Graves, 2013). Every time the recursive function of the RNN is applied and the hidden state is updated, the RNN must decide what information from the previous hidden state to store, and what information to erase, due to its limited capacity. If the RNN’s hidden representation remembers the wrong information and reaches a bad numerical state for predicting future sequence elements, likely as a result of an unexpected input, it may take many time-steps to recover.

Two proposed solutions to this problem are introducing latent variables that allow the RNN to have a probability distribution over hidden states (Chung et al., 2015b), rather than a fixed hidden state, and having RNNs with a very high memory capacity that can remember all relevant information (Graves, 2013). Introducing latent variables can result in an intractable distribution over hidden states, thus losing one of the major advantages of generative RNNs. A network with a large memory capacity could in theory reinterpret its past inputs in the context of a new surprising input. While a greater memory capacity could certainly be helpful, it is also important that the RNN has the flexibility to appropriately adjust its hidden state after viewing a surprising input, otherwise the RNN may not be able to fully take advantage of its memory capacity.

In this work, we argue that RNN architectures with flexible input-dependent transition functions should be better able to recover from surprising inputs. Our approach to making LSTMs more expressive for generative modelling combines LSTM units with a multiplicative RNN (mRNN), allowing highly flexible input dependent transitions that are easier to learn and control due to the special gating units of LSTM. Experiments compare our novel multiplicative LSTM hybrid architecture with other variants of LSTM on a range of character level language modelling tasks of varying complexity. The architecture in this work is most appropriate when it can learn parameters specifically for each possible input at a given timestep, which requires that the set of possible inputs at any timestep is tractable. Therefore, its main application is to modelling sequences of discrete mutually exclusive elements, such as language modelling and related problems.

2 Input dependent transition functions

RNNs learn a mapping from previous hidden state h_{t-1} and input x_t to hidden state h_t . Let \hat{h}_t denote the input to the next hidden state before any non-linear operation:

$$\hat{h}(t) = W_{hh}h_{t-1} + W_{hx}x_t, \quad (4)$$

where W_{hh} is the hidden to hidden matrix, and W_{hx} is the input to hidden weight matrix. For problems such as language modelling, x_t is a one-hot vector, meaning that the output of $W_{hx}x_t$ is a column in W_{hx} , corresponding to the unit element in x_t , which acts as an input dependent bias to the hidden states.

For an alphabet of N inputs and a fixed h_{t-1} , there will be N possible transition functions between h_{t-1} and \hat{h}_t . The relative magnitude of $W_{hh}h_{t-1}$ to $W_{hx}x_t$ will need to be large if we want the RNN to be able to use long range dependencies. If this is the case, the transition functions will be constrained to be similar across inputs. The resulting possible hidden state vectors will be highly correlated across the possible x_t s. This limited flexibility of input dependent transition functions therefore makes it difficult to quickly adjust the relative values of the hidden state for different inputs, and may prevent an RNN from representing complex probability distributions.

We can view the possible future hidden states in a generative RNN as a tree structure, as shown in figure 1. If the input-dependent transition functions, corresponding to the branches at each node of this tree, are closely tied and have limited flexibility, it will be more difficult for the tree to form distinct hidden representations for different sequences of inputs. However, if the generative RNN has flexible input dependent transitions, this tree will be able to grow wider more quickly, and the RNN will have the flexibility to represent more probability distributions.

In a vanilla RNN, it is difficult to allow inputs to greatly effect the hidden state vector without erasing information from the past hidden state. However, if we used an RNN architecture that allowed the

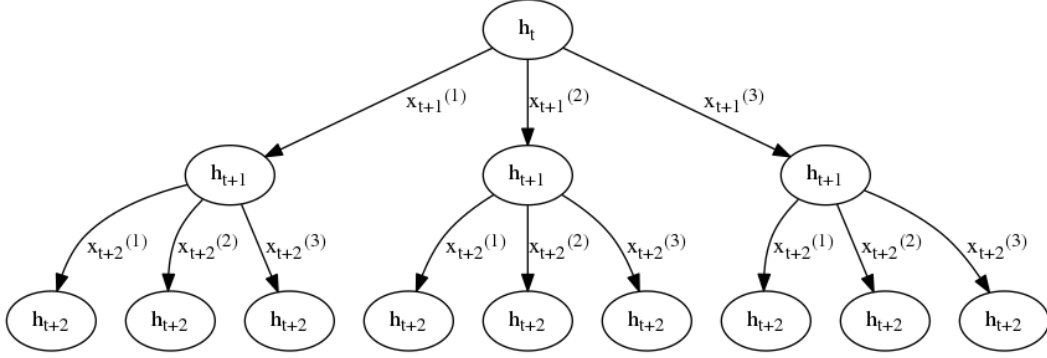


Figure 1: Diagram of hidden states of a generative RNN as a tree, where $x_t^{(n)}$ denotes which of N possible inputs is encountered at timestep t . Given h_t , the starting node of the tree, there will be a different possible h_{t+1} for every $x_{t+1}^{(n)}$. Similarly, for every h_{t+1} that can be reached from h_t , there is a different possible h_{t+2} for each $x_{t+2}^{(n)}$, and so on.

input to update the hidden state in a more interactive way, this problem could potentially be solved. For instance, if an RNN had a completely separate transition function mapping $\hat{h}_t \leftarrow h_{t-1}$ for each possible input, this would allow for the relative values of h_t to vary greatly with each possible input x_t , without having to overwrite the contribution from the previous hidden state, allowing for long term information to still be stored. This will then in turn allow the RNN to express more probability distributions. The ability to adjust to new inputs quickly without overwriting information should also make the RNN more robust to mistakes when it encounters surprising inputs, as the hidden vector is less likely to get trapped in a bad numerical state for making future predictions.

2.1 Multiplicative RNN

The multiplicative RNN (Sutskever et al., 2011) is an architecture designed specifically to allow flexible input dependent transitions. Its formulation was inspired by the tensor RNN, which is an RNN architecture that allows for a different transition matrix for each possible input. The tensor RNN features a 3-way tensor $W_{hh}^{1:N}$, which contains a separately learned transition matrix W_{hh} for each possible input in the set of inputs. The three-way tensor can be stored as an array of matrices

$$W_{hh}^{(1:N)} = \{W_{hh}^{(1)}, \dots, W_{hh}^{(N)}\}. \quad (5)$$

where superscript is used to denote the index in the array, and N the dimensionality of x_t . The specific hidden to hidden weight matrix $W_{hh}^{(x_t)}$ used for a given input x_t is then

$$W_{hh}^{(x_t)} = \sum_{n=1}^N W_{hh}^{(n)} x_t^{(n)}. \quad (6)$$

For language modelling like problems, only one unit of x_t will be on, and $W_{hh}^{(x_t)}$ will be a matrix in $W_{hh}^{(1:N)}$ corresponding to that input unit. The hidden to hidden propagation in the tensor RNN is then given by

$$\hat{h}(t) = W_{hh}^{(x_t)} h_{t-1} + W_{hx} x_t. \quad (7)$$

The tensor RNN is impractical for most problems because the number of parameters in the 3-way tensor is so high. Multiplicative RNNs (mRNNs) can be thought of as an approximation to the tensor RNN that uses parameter tying to keep the number of parameters manageable. The hidden to hidden transition matrix is created using the product of 2 dense matrices shared across inputs, with an intermediate diagonal matrix that is input dependent.

mRNNs use a factorized hidden to hidden transition matrix in place of the normal hidden to hidden matrix W_{hh} , with an input dependent intermediate diagonal matrix $\text{diag}(W_{mx} x_t)$. The input

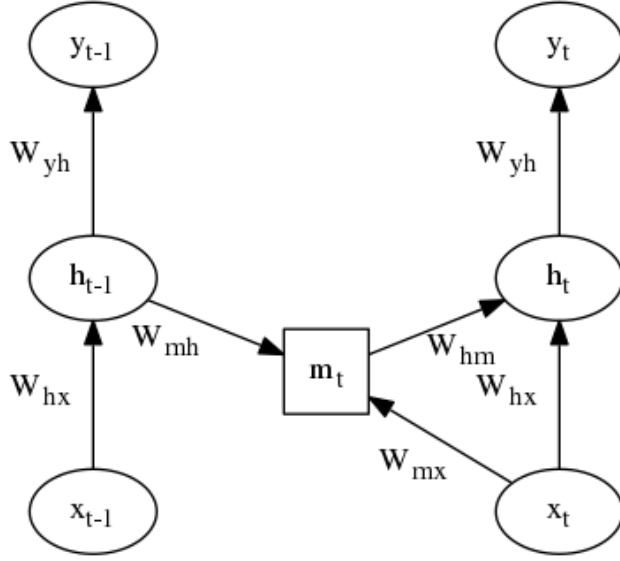


Figure 2: Diagram of a Multiplicative RNN, where edges represent propagation through weight matrices, inputs to square nodes are multiplied, and inputs to round nodes are added with non-linear functions applied.

dependent hidden to hidden weight matrix, $W_{hh}^{(x_t)}$ is then

$$W_{hh}^{(x_t)} = W_{hm} \text{diag}(W_{mx} x_t) W_{mh}. \quad (8)$$

An mRNN is then equivalent to a tensor RNN with the above formula for $W_{hh}^{(x_t)}$. For readability, an mRNN can also be described using intermediate state m_t , in which case the mapping from h_{t-1} to the \hat{h}_t is given by

$$m_t = (W_{mx} x_t) \odot (W_{mh} h_{t-1}) \quad (9)$$

$$\hat{h}_t = W_{hm} m_t + W_{hx} x_t. \quad (10)$$

The form of diagonal matrices found in mRNNs, which can be thought of as multiplicative biases, can inherently express more functions than additive biases. For instance, in ACDC neural networks (Moczulski et al., 2015), and unitary RNNs (Arjovsky et al., 2015), the learnable weights are essentially multiplicative biases, combined with special matrices that can be multiplied efficiently. The equation below shows layer to layer propagation in an ACDC network, where A and D are both diagonal matrices of learnable parameters, C is the discrete cosine transform, and C^{-1} is the inverse discrete cosine transform:

$$\hat{h}_l^T = h_{l-1}^T A C D C^{-1} \quad (11)$$

where l is the layer index. These architectures are able to learn complex functions and achieve competitive results learning only multiplicative biases, and no direct unit to unit connections. Since multiplicative biases are able to express many functions when used to scale dense matrices, having input dependent multiplicative biases is a straightforward way to efficiently allow flexibility of the RNN transition function across inputs.

Multiplicative RNNs have improved on vanilla RNNs at character level language modelling tasks (Sutskever et al., 2011; Mikolov et al., 2012), but have fallen short of the more popular LSTM architecture, for instance as shown with LSTM baselines from (Cooijmans et al., 2016). While an mRNN’s complex input dependent transitions increase its expressiveness, the standard RNN units allow no way for information to bypass these transitions. This results in the potential for difficulty in retaining information for longer periods of time.

2.2 Long short-term memory

LSTM is a commonly used RNN architecture that uses a series of multiplicative gates to control how information flows in and out of internal states of the network (Hochreiter and Schmidhuber, 1997). There are several slightly different variants of LSTM, and in this section we present the variant used in our experiments.

As in a vanilla RNN, the LSTM hidden state receives inputs from the input layer x_t and the previous hidden state h_{t-1} :

$$\hat{h}_t = W_{hx}x_t + W_{hh}h_{t-1}. \quad (12)$$

The LSTM network also has 3 gating units – input gate i , output gate o , and forget gate f – that have both recurrent and feed-forward connections:

$$i_t = \sigma(W_{ix}x_t + W_{ih}h_{t-1}) \quad (13)$$

$$o_t = \sigma(W_{ox}x_t + W_{oh}h_{t-1}) \quad (14)$$

$$f_t = \sigma(W_{fx}x_t + W_{fh}h_{t-1}), \quad (15)$$

where σ is the logistic sigmoid function. The input gate controls how much of the input to each hidden unit is written to the internal state vector c_t , and forget gates determine how much of the previous internal state c_{t-1} is preserved. This combination of write and forget gates allows the network to control what information should be stored and overwritten across each time-step. The internal state is updated by

$$c_t = f_t \odot c_{t-1} + i_t \odot \hat{h}_t. \quad (16)$$

Output gates control how much of each unit’s activation is preserved. The output gate allows the LSTM cell to keep information that is not relevant to the current output, but may be relevant later. The final output of the hidden state is given by

$$h_t = \tanh(c_t \odot o_t). \quad (17)$$

This is slightly different to the most commonly used LSTM variant, where the output gate is applied after the hyperbolic tangent.

LSTM’s extra ability to control how information is stored in each unit has proven generally useful, including for generative sequence modelling problems.

2.3 Comparing LSTM with mRNN

The LSTM and mRNN architectures both feature multiplicative units, but these units serve different purposes. In LSTM, the multiplicative gates are designed to control the flow of information through the network and store information for long periods of time. LSTM gates receive input from both the input units and hidden units, allowing multiplicative interactions between hidden units, but also potentially limiting the extent of input-hidden multiplicative interaction. LSTM’s gates are also squashed with a sigmoid, forcing them to take values between 0 and 1. This makes them more stable and easier to learn to control, but also less expressive than linear multiplicative units. The gates in LSTM give it some degree of input-dependent transition flexibility, however the main function of these gates is to give the network the ability to store information for longer periods of time.

The multiplicative units in mRNNs on the other hand are designed to allow complex input dependent transitions. They are placed in between a product of 2 dense matrices, giving more flexibility to the possible values of the final product of matrices. The effective “gates” are also linear, allowing them to express more functions. For language modelling problems, the linear multiplicative gates do not need to be controlled by the network because they are explicitly learned for each input, as the multiplicative gates for a particular input are simply a column of the weight matrix W_{mx} , which is learned directly as parameters of the network.

3 Multiplicative LSTM

Since the LSTM and mRNN architectures are complimentary, we propose multiplicative LSTM (mLSTM), a hybrid architecture that combines these two models. An RNN architecture for generative

modelling should allow flexible input dependent transitions, while also allowing information to be protected and stored across these transitions. To accomplish this, mLSTM combines the factorized hidden to hidden transition of mRNNs with the gating framework from LSTMs. The mRNN and LSTM architectures can be combined by adding connections from the mRNN’s intermediate state m_t (which is redefined below for convenience) to all of the gating units in LSTM, resulting in the following system:

$$m_t = (W_{mx}x_t) \odot (W_{mh}h_{t-1}) \quad (18)$$

$$\hat{h}_t = W_{hx}x_t + W_{hm}m_t \quad (19)$$

$$i_t = \sigma(W_{ix}x_t + W_{im}m_t) \quad (20)$$

$$o_t = \sigma(W_{ox}x_t + W_{om}m_t) \quad (21)$$

$$f_t = \sigma(W_{fx}x_t + W_{fm}m_t). \quad (22)$$

The vector m_t can have any dimensionality – we set it equal to the dimensionality of h_t for our experiments. We also chose to share m_t across all LSTM unit types. This results in a model that has only 1.25 times the number of recurrent weights as LSTM for the same number of hidden units.

The goal of this architecture is to combine the complex input dependent transition ability of an mRNN with the long time lag and information control abilities of an LSTM. Combining the gated units of LSTMs could make it easier to control (or bypass) the complex transitions in that result from the factorized hidden weight matrix. The additional sigmoid input and forget gates featured in LSTM units allow even more flexible input dependent transition functions than in regular mRNNs.

4 Related approaches

Many recent improvements to LSTM sequence modelling have used architectures that allow for more flexible input dependent transition functions. One way of doing this is by using recurrent depth, which is depth between recurrent steps. This allows a greater deal of non-linearity in the combination of inputs and previous hidden states from every time step. A previous study found recurrent depth to perform better than other kinds of non-recurrent depth for sequence modelling (Zhang et al., 2016). A concurrent study, recurrent highway networks (Zilly et al., 2016), used a more sophisticated recurrent depth that carefully controls propagation through layers using gating units. The gating units also allow for a greater deal of multiplicative interaction between the inputs and hidden units. This architecture performed very successfully on several sequence modelling tasks. Like with recurrent depth, our approach allows for a very different hidden to hidden transition function $\hat{h}_t \leftarrow h_{t-1}$ for each input; however, these functions are constrained to be linear. While adding recurrent depth could probably help our model, we believe that maximizing the input dependent flexibility of the transition function is more important for expressive sequence modelling. Recurrent depth alone can do this through non-linear layers combining hidden and input contributions, but our method can do this without depth.

Another concurrent approach which is quite similar to ours is multiplicative integration RNNs (Wu et al., 2016). These networks use Hadamard products (element wise multiplication) instead of addition when combining contributions from input and hidden units. When applying this to LSTM, this architecture was able to achieve impressive results at sequence modelling.

The main difference between multiplicative integration LSTM and multiplicative LSTM is that multiplicative LSTM applies the Hadamard product between the multiplication of two matrices. In the case of LSTM, this allows for the potential for greater expressiveness, without significantly increasing the parameters of the model (as the first matrix only has a quarter of the parameters of the second matrix).

5 Character level language modelling

One of the most common benchmarks of generative RNNs is character level language modelling. Character level language modelling requires learning long range dependencies to be successful, and utilizes far more shared parametrization than word level models. Character level language models also encompass a wide range of tasks, from basic language modelling, to more complex tasks that

include punctuation and case sensitive letters, and the most complex tasks that may feature multiple languages, or a combination of natural language and non-linguistic text.

Character level language modelling has also become of increased practical interest recently for use in machine translation (Ling et al., 2015), and in combination with word level models for general language modelling (Kim et al., 2015). Character level models model language input at the rawest level possible, giving them the most flexibility to learn feature representations. This allows them to learn subword features that can learn similar representations for semantically similar words. Using character level language models also drastically reduce the size of the output layer in a language model. Finally, character level language models can simultaneously express multiple languages as well as non-language text with full weight sharing.

6 Experiments

Our experiments compared the performance of mLSTM with regular LSTM for different character level language modelling tasks of varying complexity. Gradient computation in these experiments used truncated backpropagation through time on sequences of length 100, only resetting the hidden state every 10 thousand timesteps to allow networks to have access to information far in the past. All of these experiments used a variant of RMSprop, (Tieleman and Hinton, 2012), with normalized updates in place of a learning rate. All unnormalized update directions v_* , computed by RMSprop, were normalized to have length ℓ , where ℓ was decayed exponentially over training:

$$v \leftarrow \frac{\ell}{\sqrt{v_*^T v_*}} v_*. \quad (23)$$

We found that this allowed for fast convergence with larger batch sizes, allowing for greater parallelization during training without hurting performance.

We compared mLSTM to regular LSTM and stacked LSTM as well as other RNN character level language models reported in the literature. Stacked LSTMs were all 2-layer, and both layers contained direct connections from the inputs and to the outputs. mLSTM was benchmarked on a variety of character level language modelling tasks. The Penn Treebank dataset, (Marcus et al., 1993), was used to test small scale language modelling, and processed and raw versions of the Wikipedia text8 dataset (Hutter, 2006) were used to test large scale language modelling and byte level language modelling respectively. The European parliament dataset (Koehn, 2005) was used to examine multilingual fitting.

6.1 Penn Treebank

The Penn treebank dataset is relatively small, and consists of only case insensitive English characters, with no punctuation. It is one of the most widely used language modelling bench mark tasks. Due to its small size, the main bottleneck for performance is overfitting.

An mLSTM with 700 hidden units was fit to the Penn Treebank dataset, with no regularization other than early stopping. We used a slightly different version of this dataset, where the frequently occurring token `<unk>` was replaced by a single character, shortening the file by about 4%. To make our results comparable to other results on this dataset, we computed the total cross entropy of the test set file and divided this by the number of characters in the original test set. mLSTM achieved 1.35 bits/char test set error, compared with 1.38 bits/char for unregularized LSTM (Cooijmans et al., 2016).

6.2 Text8 dataset

Text8 contains 100 million characters of English text taken from Wikipedia in 2006. Unlike the original version of this dataset (see Hutter prize dataset experiments) the processed version of this dataset has only English text, consisting of just the 26 characters of the English alphabet plus spaces. This dataset can be found at <http://matmahoney.net/dc/textdata>. This corpus has been widely used to benchmark RNN character level language models, with the first 90 million characters used for training, the next 5 million used for validation, and the final 5 million used for testing.

architecture	test set error (bits/char)
mRNN (Mikolov et al., 2012)	1.41
multiplicative integration RNN (Wu et al., 2016)	1.39
LSTM (Cooijmans et al., 2016)	1.38
mLSTM	1.35
batch normalized LSTM (Cooijmans et al., 2016)	1.32
zoneout RNN (Krueger et al., 2016)	1.30
hierarchical multiscale LSTM (Chung et al., 2016)	1.27

Table 1: Test set error (bits/char) on Penn Treebank dataset for mLSTM compared with past work.

The first set of experiments we performed were designed to be comparable to past work that examined various ways of adding depth to RNNs. Zhang et al. (2016) benchmarked several deep LSTMs against shallow LSTMs on this dataset. The shallow LSTM had a hidden state dimensionality of 512, and the deep versions had reduced dimensionality to give them roughly the same number of parameters. Interestingly, the best reported result featured a type of LSTM with recurrent depth, which in combination with LSTM’s multiplicative gates, may have been able to learn a somewhat similar representation to the factorized hidden to hidden transition in mLSTMs. The experiment performed here with mLSTM used a hidden dimensionality of 450, giving it slightly fewer parameters than the past work. Additionally, we implemented our own LSTM baseline optimised using RMSprop with normalized updates to be consistent with our mLSTM approach. mLSTM showed an improvement over our baseline and the past work’s best deep LSTM variant, providing evidence that mLSTMs use parameters more efficiently than standard LSTM and several of its deep variants.

architecture	test set error (bits/char)
LSTM (Zhang et al., 2016)	1.65
LSTM (ours)	1.64
deep LSTM (best) (Zhang et al., 2016)	1.63
mLSTM	1.59

Table 2: Test set error (bits/char) on processed text8 dataset for mLSTM compared with shallow LSTM and the best deep LSTM variant from a past work.

We ran additional experiments to compare a large mLSTM with other reported experiments. We trained an mLSTM with hidden dimensionality of 1900 on the text8 dataset. mLSTM was able to fit the training data well and achieved a competitive performance; however it was outperformed by other architectures that are less prone to over-fitting.

architecture	test set error (bits/char)
mRNN (Mikolov et al., 2012)	1.54
multiplicative integration RNN (Wu et al., 2016)	1.52
skipping RNN (Pachitariu and Sahani, 2013)	1.48
multiplicative integration LSTM (Wu et al., 2016)	1.44
LSTM (Cooijmans et al., 2016)	1.43
mLSTM	1.40
batch normalised LSTM (Cooijmans et al., 2016)	1.36
hierarchical multiscale LSTM (Chung et al., 2016)	1.30

Table 3: Text8 dataset test set error in bits/char.

6.3 Hutter-prize dataset

These experiments featured the raw version of the Wikipedia dataset, which was originally used for the Hutter Prize compression benchmark (Hutter, 2006). This dataset consists mostly of English language text and mark-up language text, but also contains text in other languages, including non-Latin languages. The dataset is modelled using a utf-8 encoding, and contains 205 unique bytes.

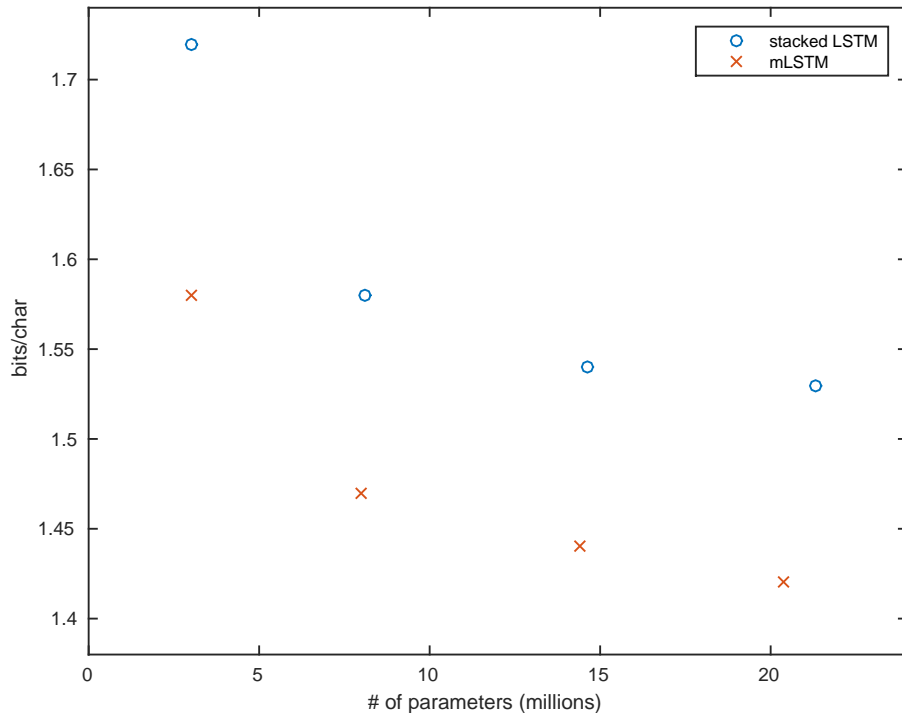


Figure 3: Hutter prize validation performance in bits/char plotted against number of network parameters for mLSTM and stacked LSTM.

The task of modelling this dataset could be described as byte level language modelling rather than character level language modelling, as certain characters in this dataset are represented by multiple bytes.

We compared mLSTMs and 2-layer stacked LSTMs for varying network sizes, ranging from about 3–20 million parameters. These results all used RMS prop with normalized updates, stopping after 4 epochs. The results, given in figure 3, show that mLSTM gives a modest improvement across all network sizes. For the network sizes tested, mLSTMs were able to perform equivalently to 2-layer stacked LSTMs with more than twice as many parameters.

We hypothesized that mLSTM’s superior performance over stacked LSTM was in part due to its ability to recover from surprising inputs. To test this we looked specifically at each network’s performance after viewing surprising inputs that occurred naturally in the validation set. We sorted all characters in the validation set by the average loss taken by the largest mLSTM and the largest stacked LSTM, and considered the 10% with the largest average loss. Both networks performs roughly equally on this set of surprising characters, with mLSTM and stacked LSTM taking losses of 6.27 bits/character and 6.29 bits/character respectively. However, stacked LSTM tended to take much larger losses than mLSTM in the timesteps immediately preceding surprising inputs. One to four time-steps after a surprising input occurred, mLSTM and stacked LSTM took average losses of (2.26, 2.04, 1.61, 1.51) and (2.48, 2.25, 1.79, 1.67) bits per character respectively, as shown in figure 4. mLSTM’s overall advantage over stacked LSTM was 1.42 bits/char to 1.53 bits/char, so mLSTM’s advantage over stacked LSTM is greater after a surprising input than it is in general. This supports our hypothesis that mLSTM’s flexible input dependent transitions help it recover from surprising inputs.

We also ran an additional experiment with our largest mLSTM and stacked LSTM models using dynamic evaluation, where the networks weights are adapted to fit recent sequences, following (Graves, 2013). Dynamic evaluation can be very helpful for non-uniform sequences that contain local regularities. An RNN with fixed weights may underfit because it is unable to simultaneously model

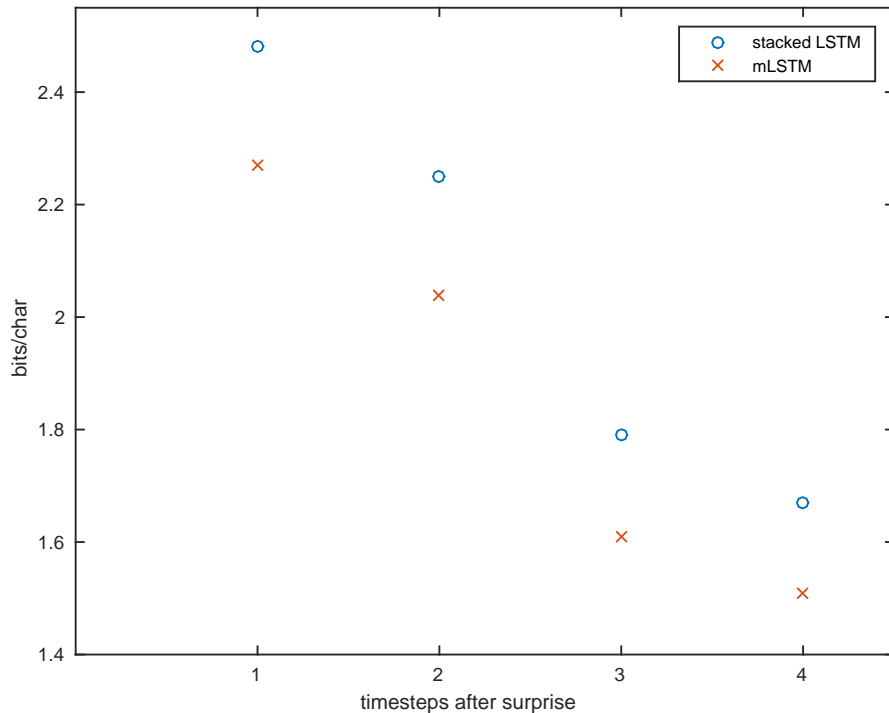


Figure 4: Cross entropy loss for mLSTM and stacked LSTM immediately proceeding a surprising input.

all the modes of data. Additionally, a model may incur a significant validation error if it comes across a new mode of sequence it has never seen before. Dynamically adapting to recent sequences can be beneficial as it helps reduce both of these problems.

For dynamic evaluation, we divided the validation set into sequences of length 50 in order of occurrence, and used online learning to adjust to these sequences. After predicting a sequence and incurring a loss, we trained the RNN for a single iteration on that sequence, using RMSprop (with a learning rate, as normalized updates no longer make sense in a stochastic setting). After updating the RNN, we recomputed the forward pass through this sequence to update the final hidden state. The updated RNN was then used to predict the next sequence of 50 elements, and this process was repeated. A decay rate penalized large deviations from the RNN’s original parameters to prevent the adapted RNN from drifting too far from the original RNN. We explored two kinds of dynamic evaluation: one where all the parameters were dynamically adapted (tested for both mLSTM and stacked LSTM), and one where only bias parameters were adapted (only tested for mLSTM). Dynamically adapting all parameters becomes very memory intensive to parallelize, because it requires a separate parameter vector to be stored for each parallel thread. However the number of bias parameters is quadratically fewer than the total number of parameters, so bias-only dynamic adaptation can be much faster to run with limited GPU resources.

mLSTM and stacked LSTM both achieved much better validation errors using dynamic evaluation, with mLSTM’s dynamic cross entropy error dropping slightly more (relative to its static error) than our stacked LSTM baseline. mLSTM’s static result was already much better than stacked LSTM’s, and improving an already strong model should be more difficult than improving a weaker one. This result suggests that mLSTM may be more suitable for dynamic evaluation than stacked LSTM. This could be the case if mLSTM’s extra expressiveness allows it to adapt to new sequences online more quickly. However, a previous approach to stacked LSTM with dynamic evaluation, (Graves, 2013), saw a much larger improvement from dynamic evaluation, performing much weaker than our stacked LSTM when evaluated statically, and equivalently when evaluated dynamically. The previous work

may have gained a dynamic evaluation advantage by using 7 layers, may have used a better dynamic fitting algorithm, and had an easier task of improving a weaker model. mLSTM’s result with dynamic evaluation seems quite strong, however, there are currently no competitive baselines that we are aware of to compare it against.

The best results for stacked LSTM and mLSTM are given in table 4, alongside results from the literature. mLSTM performs at near state of the art level when evaluated statically, and greatly outperforms the best static models and other dynamic models when evaluated dynamically.

architecture	validation error (bits/char)
7-layer stacked LSTM (Graves, 2013)	1.67
gf-LSTM (Chung et al., 2015a)	1.58
2-layer stacked LSTM (ours)	1.53
grid LSTM (Kalchbrenner et al., 2015)	1.48
multiplicative integration LSTM (Wu et al., 2016)	1.44
mLSTM	1.42
recurrent highway networks (Zilly et al., 2016)	1.42
hierarchical multiscale RNNs (Chung et al., 2016)	1.40
recurrent memory array structures (Rocki, 2016a)	1.40
feedback LSTM (Rocki, 2016b)	1.39*
feedback LSTM + zoneout (Rocki, 2016b)	1.37*
mLSTM (dynamic, bias only)	1.34*
7-layer stacked LSTM (dynamic) (Graves, 2013)	1.33*
2-layer stacked LSTM (ours, dynamic)	1.33*
mLSTM (dynamic)	1.20*

Table 4: Raw wikipedia dataset validation error in bits/char. Results labelled with * use the error signal to update the hidden state, and architectures labelled with (dynamic) use gradient descent based fitting to recent sequences to perform this adjustment.

6.4 Multilingual learning

Fitting a character level RNN to model multiple languages using shared parameters is another interesting metric of architectural expressiveness. These experiments compared the relative ability of LSTM and mLSTM at fitting a dataset in a single language and a combined dataset with two separate languages. The first 100 million characters of the English and Spanish translations of the European parliament dataset were used to make an English dataset and a Spanish dataset. Each dataset was split 90-5-5 for training, validation, and testing. A third Spanish-English hybrid dataset was created by combining the Spanish and English datasets, resulting in a dataset twice as large. The hybrid dataset used a 180-10-10 million character split for training, validation, and testing, corresponding to the same datasets as used in the English and Spanish only versions. These datasets were left in their raw form, containing punctuation and both upper-case and lower-case letters. The LSTMs in these experiments had a hidden dimensionality of 2200, and the mLSTMs had a hidden dimensionality of 1900. All experiments were run for 4 epochs.

architecture	English only test error	Spanish only test error	Spanish-English test error
LSTM	1.13	1.01	1.14
mLSTM	1.05	0.95	1.04

Table 5: European parliament test error in bits/char for LSTM and mLSTM on English only, Spanish only, and mixed English-Spanish.

The mLSTM generally seemed to outperform LSTM at this task. However, there also seemed to be an interaction effect with the number of languages. Increasing the complexity of the task by forcing the RNN to learn 2 languages instead of 1 presented a larger fitting difficulty for the LSTM than the mLSTM. This suggests that mLSTM was better able to scale with the more complex task of fitting a larger dataset with two languages.

7 Discussion

This work combined the mRNN’s factorized hidden weights with the LSTM’s hidden units for generative modelling of discrete multinomial sequences. This architecture was motivated by its ability to have both controlled and flexible input dependent transitions, which allows for fast changes to the distributed hidden representation without erasing information. In a series of character level language modelling experiments, mLSTM showed improvements over LSTM and its deep variants at a variety of character level language modelling tasks. This relative improvement increased with the complexity of the task, and provided evidence that mLSTM has a more powerful fitting ability for character level language modelling than regular LSTM and its common deep variants. mLSTM performed competitively at large scale character level language modelling, and achieved a state of the art result of 1.20 bits/character on the Hutter prize dataset when combined with dynamic evaluation.

While these results are promising, it remains to be seen how mLSTM performs at word level language modelling and other discrete multinomial generative modelling tasks, and whether mLSTM can be formulated to apply more broadly to tasks with continuous or non-sparse input units. We also hope this work will motivate further exploration in generative RNN architectures with flexible input dependent transition functions.

References

- Arjovsky, M., Shah, A., and Bengio, Y. (2015). Unitary evolution recurrent neural networks. *arXiv preprint arXiv:1511.06464*.
- Bengio, Y., Simard, P., and Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *Neural Networks, IEEE Transactions on*, 5(2):157–166.
- Chung, J., Ahn, S., and Bengio, Y. (2016). Hierarchical multiscale recurrent neural networks. *arXiv preprint arXiv:1609.01704*.
- Chung, J., Gülçehre, C., Cho, K., and Bengio, Y. (2015a). Gated feedback recurrent neural networks. *CoRR*, abs/1502.02367.
- Chung, J., Kastner, K., Dinh, L., Goel, K., Courville, A. C., and Bengio, Y. (2015b). A recurrent latent variable model for sequential data. In *Advances in neural information processing systems*, pages 2962–2970.
- Cooijmans, T., Ballas, N., Laurent, C., and Courville, A. (2016). Recurrent batch normalization. *arXiv preprint arXiv:1603.09025*.
- Graves, A. (2013). Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*.
- Hochreiter, S., Bengio, Y., Frasconi, P., and Schmidhuber, J. (2001). Gradient flow in recurrent nets: The difficulty of learning long-term dependencies. In Kremer and Kolen, editors, *A Field Guide to Dynamical Recurrent Neural Networks*. IEEE Press.
- Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, 9:1735–1780.
- Hutter, M. (2006). The human knowledge compression prize.
- Kalchbrenner, N., Danihelka, I., and Graves, A. (2015). Grid long short-term memory. *arXiv preprint arXiv:1507.01526*.
- Kim, Y., Jernite, Y., Sontag, D., and Rush, A. M. (2015). Character-aware neural language models. *arXiv preprint arXiv:1508.06615*.
- Koehn, P. (2005). Europarl: A parallel corpus for statistical machine translation. In *MT Summit*, volume 5, pages 79–86.
- Krueger, D., Maharaj, T., Kramár, J., Pezeshki, M., Ballas, N., Ke, N. R., Goyal, A., Bengio, Y., Larochelle, H., Courville, A., et al. (2016). Zoneout: Regularizing RNNs by randomly preserving hidden activations. *arXiv preprint arXiv:1606.01305*.
- Ling, W., Trancoso, I., Dyer, C., and Black, A. W. (2015). Character-based neural machine translation. *arXiv preprint arXiv:1511.04586*.

- Marcus, M. P., Marcinkiewicz, M. A., and Santorini, B. (1993). Building a large annotated corpus of English: The Penn Treebank. *Computational linguistics*, 19(2):313–330.
- Mikolov, T., Sutskever, I., Deoras, A., Le, H.-S., Kombrink, S., and Cernocky, J. (2012). Subword language modeling with neural networks. *preprint (http://www.fit.vutbr.cz/imikolov/rnnlm/char.pdf)*.
- Moczulski, M., Denil, M., Appleyard, J., and de Freitas, N. (2015). ACDC: A structured efficient linear layer. *arXiv preprint arXiv:1511.05946*.
- Pachitariu, M. and Sahani, M. (2013). Regularization and nonlinearities for neural language models: when are they needed? *arXiv preprint arXiv:1301.5650*.
- Rocki, K. (2016a). Recurrent memory array structures. *arXiv preprint arXiv:1607.03085*.
- Rocki, K. M. (2016b). Surprisal-driven feedback in recurrent networks. *arXiv preprint arXiv:1608.06027*.
- Sutskever, I., Martens, J., and Hinton, G. E. (2011). Generating text with recurrent neural networks. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 1017–1024.
- Tieleman, T. and Hinton, G. (2012). Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural Networks for Machine Learning*, 4(2).
- Wu, Y., Zhang, S., Zhang, Y., Bengio, Y., and Salakhutdinov, R. (2016). On multiplicative integration with recurrent neural networks. *arXiv preprint arXiv:1606.06630*.
- Zhang, S., Wu, Y., Che, T., Lin, Z., Memisevic, R., Salakhutdinov, R., and Bengio, Y. (2016). Architectural complexity measures of recurrent neural networks. *arXiv preprint arXiv:1602.08210*.
- Zilly, J. G., Srivastava, R. K., Koutník, J., and Schmidhuber, J. (2016). Recurrent highway networks. *arXiv preprint arXiv:1607.03474*.